SPI Interface Project

“SPI Slave with Single Port Ram”

Verilog implementation and FPGA flow

BY: Youssif Ahmed Sayed

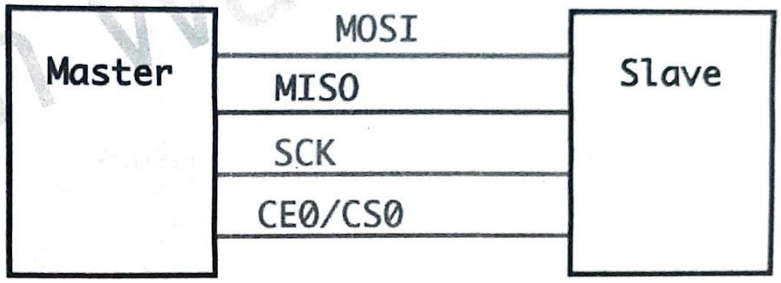
Table of Contents

1. Project Overview

1.1. Introduction

This project involves the design and implementation of a digital communication system using the Serial Peripheral Interface (SPI) protocol. The system consists of an SPI Slave module, a Single Port RAM, and an SPI Wrapper module that integrates these components. The implementation was carried out on an FPGA platform using Xilinx Vivado, demonstrating the design's functionality and performance.

1.2. SPI Interface Diagram

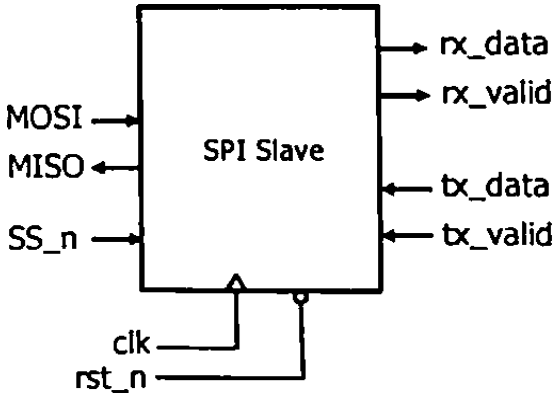


- **Master:** The master device initiates the communication and controls the data transfer. It generates the clock signal (**SCK**) and selects the slave device by activating the chip select line **(CE0/CS0).** The master sends data to the slave through the **MOSI** (Master Out Slave In) line and receives data from the slave through the **MISO** (Master in Slave Out) line.

- **Slave:** The slave device responds to the master's commands. It receives data from the master via the **MOSI** line and sends data back to the master through the **MISO** line. The slave device is selected for communication when the **CE0/CS0** line is active (usually low).

1.3. Project Specifications

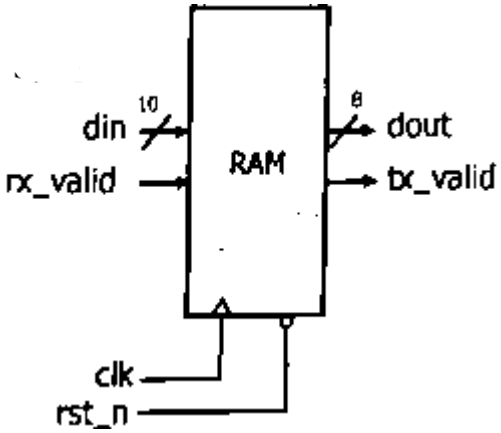
1.3.1. SPI Slave Interface



It handles the SPI protocol, converting serial data to parallel for storage in the RAM and vice versa. This block interfaces with the master device.

Name	Type	Size	Description
MOSI	Input	1 bit	**Master output slave input signal**
SS_n		1 bit	**Active low Slave select signal**
clk		1 bit	**Clock Signal**
arst_n		1 bit	**Active low asynchronous reset signal**
tx_data		8 bits	**Transmitted data required from the RAM to the Master**
tx_valid		1 bit	**HIGH only when the data is ready to be received from RAM**
rx_data	Output	10 bits	**Received data from the Master converted from serial into parallel to the RAM**
rx_valid		1 bit	**HIGH only when the data is ready to be sent to RAM**
MISO		**1 bit**	**Master input slave output signal**

1.3.2. Single Port Ram



It acts as the storage medium where the SPI Slave stores the received data and retrieves it upon request.

- Memory Depth: 256 words.
- Address Size: 8 bits.
- Data Width: 8-bit data stored and retrieved.
- Control Signals: rx_valid for writing data, tx_valid for reading data.

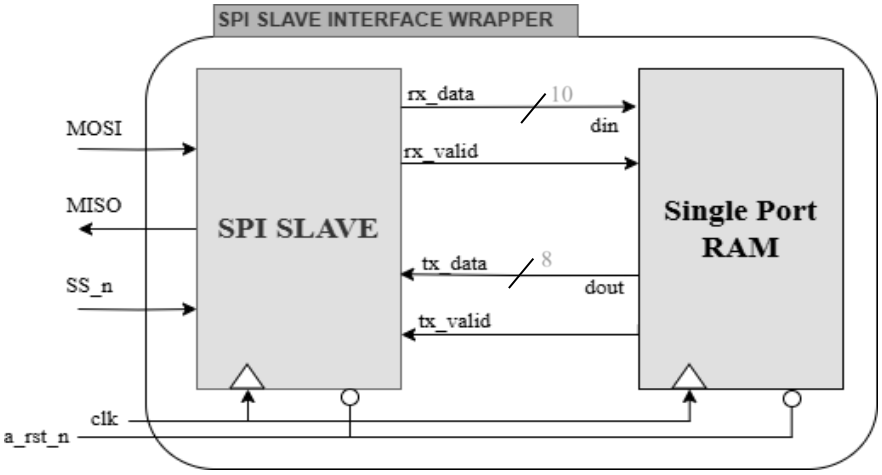
Parameters	
MEM_DEPTH	Default: 256
ADDR_SIZE	Default: 8

Name	Type	Size	Description
clk	Input	1 bit	Clock Signal
arst_n		1 bit	Active low asynchronous reset signal
din		10 bits	Data Input
rx_valid		1 bit	HIGH only accept din[7:0] to save the write/read address internally or write a memory word depending on the most significant 2 bits din[9:8]
dout	Output	8 bits	Data Output
tx_valid		1 bit	Whenever the command is memory read the tx_valid should be HIGH

- Din[9:8] selects the mode for Read/ Write on the single port asynchronous RAM

din[9:8]	Command	Description
00	Write	Hold din[7:0] internally as write address
01		Write din[7:0] in the memory with write address held previously
10	Read	Hold din[7:0] internally as read address
11		Read the memory with read address held previously, tx_valid should be HIGH, dout holds the word read from the memory, ignore din[7:0].

1.3.3. SPI Wrapper



It serves as the top-level module, managing the data flow between the SPI Slave and the RAM. It connects the SPI Slave's outputs (like rx_data and rx_valid) to the RAM's inputs (din, rx_valid), and the RAM's outputs (dout, tx_valid) to the SPI Slave's inputs (tx_data, tx_valid). It integrates the SPI Slave and RAM, with external connections for MOSI, MISO, SS_n, clk, and arst_n.

1.4. Design Flow

1. **Design Entry**: Verilog HDL was used to code the SPI Slave, Single Port RAM, and SPI Wrapper modules.
2. **Synthesis**: The design was synthesized in Xilinx Vivado, generating the netlist for FPGA implementation.
3. **Implementation**: Placement and routing were performed in Vivado, with timing constraints applied and verified.
4. **Simulation**: Functional simulations were carried out using testbenches, focusing on verifying SPI operations and memory interactions.
5. **Testing**: The system was tested on FPGA, validating the data integrity and timing requirements.

1.5. Summary

This project demonstrates the successful implementation of an SPI communication system on an FPGA. The SPI Slave module effectively interfaces with the master device, while the Single Port RAM stores and retrieves data as required. The SPI Wrapper integrates these components, ensuring smooth data transfer. The project was fully synthesized, implemented, and tested using Xilinx Vivado, with results meeting the design specifications.

3. RTL Design

3.1. Single Port Ram module

```
module single_port_Ram #(
  /*------------Parameters------------*/
    parameter MEM_DEPTH = 256,         /*  Memory depth */
    parameter ADD_SIZE = 8            /* Address size based upon the memory depth */
)(
  /*--------------Inputs--------------*/
  input  [9:0]din,    /* Data input, din[9] detemines write or read, 0 => Write, 1 => read
                       * din[9:8] = 00 => Write, Hold din[7:0] internally as write address
                       * din[9:8] = 01 => Write, write din[7:0] in the memory with wr address held previously
                       * din[9:8] = 10 => Read, Hold din[7:0] internally as Read address
                       * din[9:8] = 01 => Read the memory with rd address held previously,tx_valid = HIGH,
                       *                  dout holds the word read from the memory, ignore din[7:0]     */
  input  clk,         /*  clock signal input */
  input  arst_n,      /*  active low asynchronous reset */
  input  rx_valid,    /*  if HIGH: accept din[7:0] to save the wr/rd address internally or write a memory word */
  /*-------------outputs-------------*/
  output reg [7:0]dout,           /* data out of Ram */
  output reg tx_valid             /*  Whenever the command is memory read, the tx_valid should be HIGH */
);
  /* internal bus to hold the address internally */
  reg [ADD_SIZE-1:0]addr_internal;

  /* memory declaration */
  (* ram_style = "block" *)reg [7:0]mem[MEM_DEPTH-1:0];


  always @(posedge clk) begin
    if(~arst_n)begin
      dout <= 0;
      tx_valid <= 0;
    end else if(rx_valid) begin
      case (din[9:8])
          2'b00 :
            /* Write operation - hold the write address */
            addr_internal <= din[7:0];
          2'b01 :
            /* Write operation - write data to memory in the internal address held previously  */
            mem[addr_internal] <= din[7:0];
          2'b10 :
            /* Read operation - hold the read address */
            addr_internal <= din[7:0];
          2'b11 : begin
            /* Read operation - read data from memory mem[addr_internal]  */
            dout <= mem[addr_internal];
            tx_valid <= 1;
          end
          default: begin
            /* deafult case  */
            dout <= 0;
            tx_valid <= 0;
          end
      endcase
    end else begin
      /* reset tx_valid when the rx_valid is low */
      tx_valid <= 0;
    end
  end

endmodule
```

3.2. SPI Slave Interface “Sequential Encoding”

Sequential encoding has the highest setup slack time after implementation.

```
module SPI_Slave (
  /*--------------Inputs--------------*/
    input  MOSI,        /* the serial date sent from the master */
    input  SS_n,        /* start and end communication from master side */
    input  [7:0]tx_data,/* the data to write in the memory */
    input  tx_valid,    /* the signal dedicate that tx_data is ready to covert from parallel to serial by slave*/
    input  clk,         /*  clock signal input */
    input  arst_n,      /*  active low synchronous reset */
  /*--------------outputs-------------*/
    output reg MISO,         /* the serial data sent to the master */
    output reg [9:0]rx_data, /* the data which is read from the memory */
    output reg rx_valid      /* the signal dedicates that rx_data coverted to parallel by slave and ready for memory */
);
  /*------FSM States Declaration------*/
    localparam IDLE       = 3'b000;
    localparam CHX_CMD    = 3'b001;
    localparam WRITE      = 3'b010;
    localparam READ_ADD   = 3'b011;
    localparam READ_DATA  = 3'b100;

  /* Xilinx Vivado's Attribute FSM Encoding Method (Gray, One_Hot, Sequential) */
    (* fsm_encoding = "sequential" *)
  /*--------internal signals----------*/
    reg [2:0]CS,NS;         /* Current and Next States */
    reg [3:0]rx_counter;    /* to access the rx_data bus (8-bit) during converting from serial to parallel */
    reg [3:0]tx_counter;    /* to access the tx_data bus (8-bit) during converting from serial to parallel */
    reg rd_addr_hold;       /* Hold read address */
  /*------------State memory----------*/
  always @(posedge clk ) begin
    if(~arst_n)begin
        CS <= IDLE;
    end else
        CS <= NS;
  end
  /*----------Next State Logic--------*/
  always @(*) begin
    case (CS)
        IDLE :begin
            if(SS_n)
                NS = IDLE;
            else
                NS = CHX_CMD;
        end
        CHX_CMD : begin
            /* the master opens the communication to the slave */
            if(~SS_n) begin
                /* if MOSI is low, operation will be write */
                if(~MOSI)
                    NS = WRITE;
                /* if MOSI is high, operation will be read */
                else begin
                    /* if the read addr is held, the next is read the data */
                    if(rd_addr_hold)
                        NS = READ_DATA;
                    /* if the read addr isn't held, the next is read address */
                    else
                        NS = READ_ADD;
                end
            end
            /* the master ends the communication to the slave */
            else
                NS = IDLE;
        end
        WRITE : begin
            /* the master opens the communication to the slave */
            if(~SS_n) begin
                NS = WRITE;
            end
            /* the master ends the communication to the slave */
            else
                NS = IDLE;
        end
```

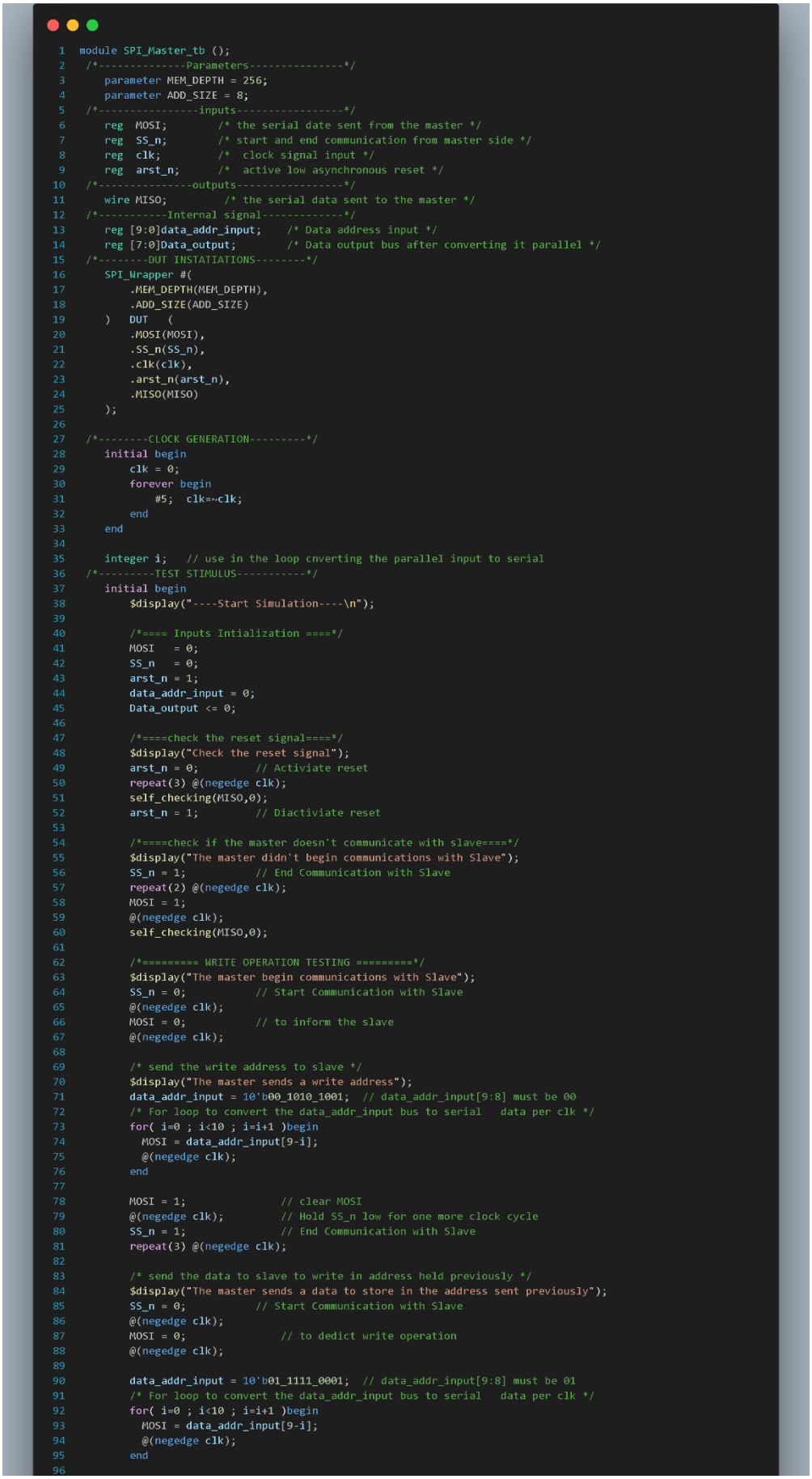
```
        READ_ADD : begin
            /* the master opens the communication to the slave */
            if(~SS_n) begin
                NS = READ_ADD;
            end
            /* the master ends the communication to the slave */
            else
                NS = IDLE;
        end
        READ_DATA : begin
            /* the master opens the communication to the slave */
            if(~SS_n)
                NS = READ_DATA;
            /* the master ends the communication to the slave */
            else
                NS = IDLE;
        end
        default:  NS = IDLE;
    endcase
  end
  /*----------Output logic----------*/
  always @(posedge clk ) begin
    if(~arst_n)begin
        MISO             <= 0;
        rx_data          <= 0;
        rx_valid         <= 0;
        rd_addr_hold     <= 0;
        rx_counter       <= 0;
        tx_counter       <= 0;
    end
    case (CS)
        IDLE :begin
            MISO     <= 0;
            rx_data  <= 0;
            rx_valid <= 0;
            rx_counter  <= 0;
            tx_counter  <= 0;
        end
        CHX_CMD:begin
            MISO     <= 0;
            rx_data  <= 0;
            rx_valid <= 0;
            rx_counter  <= 0;
            tx_counter  <= 0;
        end
        WRITE:begin
            if(rx_counter<10)begin
                rx_data[9-rx_counter] <= MOSI ;
                rx_counter <= rx_counter + 1;
                rx_valid <= 0;
            end else begin
                rx_valid <= 1;
                rx_counter <= 0;
            end
        end
        READ_ADD:begin
            if(rx_counter<10)begin
                rx_data[9-rx_counter] <= MOSI ;
                rx_counter <= rx_counter + 1;
            end else begin
                rx_valid <= 1;
                rd_addr_hold <=1;
                rx_counter <= 0;
            end
        end
        READ_DATA:begin
            if(rx_counter<10)begin
                rx_data[9-rx_counter] <= MOSI ;
                rx_counter <= rx_counter + 1;
            end else begin
                rx_valid <= 1;
                /* Convert the read data from parallel to serial */
                if(tx_valid)begin
                    if(tx_counter<10)begin
                        MISO <= tx_data[tx_counter] ;
                        tx_counter <= tx_counter + 1;
                    end else begin
                        MISO <= 0;
                        rx_counter   <= 0;
                        tx_counter   <= 0;
                        rd_addr_hold <= 0;
                        rx_valid <= 0;
                    end
                end
            end
        end
```

```
        default:begin
          MISO <= 0;
          rx_data <= 0;
          rx_valid <= 0;
          tx_counter <=0;
          rx_counter <=0;
        end
      endcase
    end

endmodule //SPI_Slave
```

3.3. SPI Wrapper module

```
module SPI_Wrapper #(
    /*------------Parameters------------*/
        parameter MEM_DEPTH = 256,         /*  Memory depth */
        parameter ADD_SIZE = 8             /* Address size based upon the memory depth */
    )(
    /*--------------Inputs--------------*/
        input  MOSI,        /* the serial date sent from the master */
        input  SS_n,        /* start and end communication from master side */
        input  clk,         /*  clock signal input */
        input  arst_n,      /*  active low asynchronous reset */
    /*--------------outputs-------------*/
        output MISO         /* the serial data sent to the master */
    );
    /*--------internal signals which connect Ram with SPI Slave----------*/
    wire [7:0]tx_data;    /* connect output of ram "dout" with input of Slave "tx_data" */
    wire tx_valid;        /* connect output of ram "tx_valid" with input of Slave "tx_valid" */
    wire [9:0]rx_data;    /* connect input of ram "din" with output of Slave "rx_data" */
    wire rx_valid;        /* connect input of ram "rx_valid" with output of Slave "rx_valid" */

    /*-------SPI_Slave Instantiation------*/
    SPI_Slave SPI_slave (
                .MOSI(MOSI),         /* the serial date sent from the master */
                .SS_n(SS_n),         /* start and end communication from master side */
                .tx_data(tx_data),   /* the data to write in the memory */
                .tx_valid(tx_valid), /* the signal dedicate that tx_data is ready to covert from parallel to serial by slave*/
                .clk(clk),           /*  clock signal input */
                .arst_n(arst_n),     /*  active low synchronous reset */
                .MISO(MISO),         /* the serial data sent to the master */
                .rx_data(rx_data),   /* the data which is read from the memory */
                .rx_valid(rx_valid)  /* the signal dedicates that rx_data coverted to parallel by slave and ready for memory */
                );

    /*---------Ram Instantiation--------*/
    single_port_Ram #(
                .MEM_DEPTH(MEM_DEPTH),
                .ADD_SIZE(ADD_SIZE))
                RAM (
                .din(rx_data),          /*  Ram Data input */
                .clk(clk),              /*  clock signal input */
                .arst_n(arst_n),        /*  active low asynchronous reset */
                .rx_valid(rx_valid),    /*  informs ram that data input is valid */
                .dout(tx_data),         /*  Ram Data output */
                .tx_valid(tx_valid)     /*  dedicated that data output is valid */
                );

endmodule //SPI_Wrapper

```

4. SPI Wrapper Testbench as Master



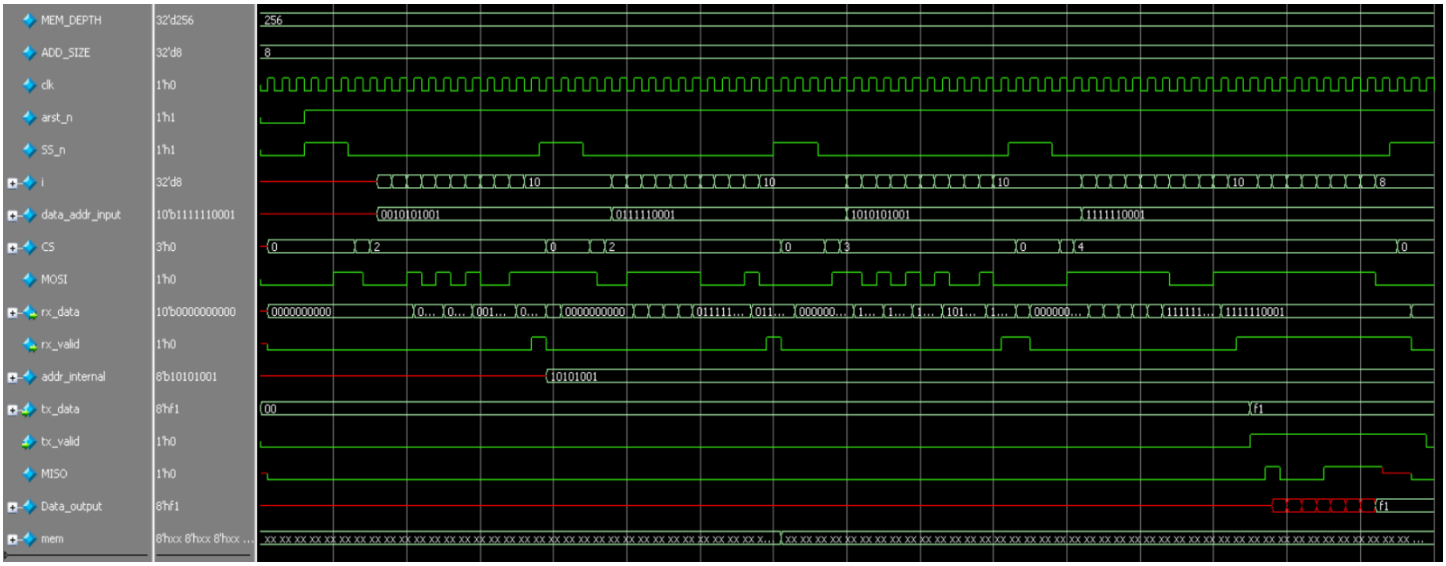
```
module SPI_Master_tb ();
 /*-------------Parameters--------------*/
    parameter MEM_DEPTH = 256;
    parameter ADD_SIZE = 8;
 /*----------------inputs----------------*/
    reg  MOSI;        /* the serial date sent from the master */
    reg  SS_n;        /* start and end communication from master side */
    reg  clk;         /*  clock signal input */
    reg  arst_n;      /*  active low asynchronous reset */
 /*---------------outputs----------------*/
    wire MISO;         /* the serial data sent to the master */
 /*----------Internal signal------------*/
    reg [9:0]data_addr_input;    /* Data address input */
    reg [7:0]Data_output;        /* Data output bus after converting it parallel */
 /*--------DUT INSTATIATIONS--------*/
    SPI_Wrapper #(
        .MEM_DEPTH(MEM_DEPTH),
        .ADD_SIZE(ADD_SIZE)
    )   DUT   (
        .MOSI(MOSI),
        .SS_n(SS_n),
        .clk(clk),
        .arst_n(arst_n),
        .MISO(MISO)
    );

 /*--------CLOCK GENERATION---------*/
    initial begin
        clk = 0;
        forever begin
            #5;  clk=~clk;
        end
    end

    integer i;   // use in the loop cnverting the parallel input to serial
 /*---------TEST STIMULUS-----------*/
    initial begin
        $display("----Start Simulation----\n");

        /*==== Inputs Intialization ====*/
        MOSI   = 0;
        SS_n   = 0;
        arst_n = 1;
        data_addr_input = 0;
        Data_output <= 0;

        /*====check the reset signal====*/
        $display("Check the reset signal");
        arst_n = 0;          // Activiate reset
        repeat(3) @(negedge clk);
        self_checking(MISO,0);
        arst_n = 1;          // Diactiviate reset

        /*====check if the master doesn't communicate with slave====*/
        $display("The master didn't begin communications with Slave");
        SS_n = 1;            // End Communication with Slave
        repeat(2) @(negedge clk);
        MOSI = 1;
        @(negedge clk);
        self_checking(MISO,0);

        /*========= WRITE OPERATION TESTING =========*/
        $display("The master begin communications with Slave");
        SS_n = 0;            // Start Communication with Slave
        @(negedge clk);
        MOSI = 0;            // to inform the slave
        @(negedge clk);

        /* send the write address to slave */
        $display("The master sends a write address");
        data_addr_input = 10'b00_1010_1001;  // data_addr_input[9:8] must be 00
        /* For loop to convert the data_addr_input bus to serial   data per clk */
        for( i=0 ; i<10 ; i=i+1 )begin
          MOSI = data_addr_input[9-i];
          @(negedge clk);
        end

        MOSI = 1;                // clear MOSI
        @(negedge clk);          // Hold SS_n low for one more clock cycle
        SS_n = 1;                // End Communication with Slave
        repeat(3) @(negedge clk);

        /* send the data to slave to write in address held previously */
        $display("The master sends a data to store in the address sent previously");
        SS_n = 0;            // Start Communication with Slave
        @(negedge clk);
        MOSI = 0;                // to dedict write operation
        @(negedge clk);

        data_addr_input = 10'b01_1111_0001;  // data_addr_input[9:8] must be 01
        /* For loop to convert the data_addr_input bus to serial   data per clk */
        for( i=0 ; i<10 ; i=i+1 )begin
          MOSI = data_addr_input[9-i];
          @(negedge clk);
        end

```

```
        MOSI = 0;               // clear MOSI
        @(negedge clk);         // Hold SS_n low for one more clock cycle
        SS_n = 1;               // End Communication with Slave
        repeat(3) @(negedge clk);

        $display("\nCheck address '169' in the memory. It should have data '1111_0001'= 'F1' ");
        self_checking_8bit(DUT.RAM.mem[169],8'hF1);

        /*========= Read OPERATION TESTING ==========*/
        $display("The master begin communications with Slave");
        SS_n = 0;           // Start Communication with Slave
        @(negedge clk);
        MOSI = 1;           // to inform the slave there is read operation
        @(negedge clk);
        /* send the write address to slave */
        $display("The master sends a read address");
        data_addr_input = 10'b10_1010_1001;  // data_addr_input[9:8] must be 10
        /* For loop to convert the data_addr_input bus to serial data per clk */
        for( i=0 ; i<10 ; i=i+1 )begin
          MOSI = data_addr_input[9-i];
          @(negedge clk);
        end

        MOSI = 0;               // clear MOSI
        @(negedge clk);         // Hold SS_n low for one more clock cycle
        SS_n = 1;               // End Communication with Slave
        repeat(3) @(negedge clk);

        /* send the data bits code to slave to read from address held previously */
        $display("The master sends a data code bits to read from the address sent previously");
        SS_n = 0;           // Start Communication with Slave
        @(negedge clk);
        MOSI = 1;           // to dedict write operation
        @(negedge clk);

        data_addr_input = 10'b11_1111_0001;  // data_addr_input[9:8] must be 11 and other bits are dummy
        /* For loop to convert the data_addr_input bus to serial data per clk */
        for( i=0 ; i<10 ; i=i+1 )begin
          MOSI = data_addr_input[9-i];
          @(negedge clk);
        end

        $display("\nCheck address '169' in the memory. and the MISO bits transmitted");

        @(negedge clk);        // Ensure data is stable
        @(negedge clk);        // to update tx_data

        /* For loop to convert the MISO from serial data per clk to the internal signal 'Data_output' bus */
        for(i=0; i<8; i=i+1) begin
            @(negedge clk);
            Data_output[i] = MISO;
        end

        MOSI = 0;               // clear MOSI
        @(negedge clk);         // Hold SS_n low for one more clock cycle
        SS_n = 1;               // End Communication with Slave
        repeat(3) @(negedge clk);
        self_checking_8bit(Data_output,DUT.RAM.mem[169]);

        $display("----End Simulation----\n");
        $stop;

    end

    /*--------Self_Checking 1-bit---------*/
       task self_checking;
            input DUT_out;
            input expected_tb;
            begin
                // Check if the output is correct
                if (DUT_out == expected_tb) begin
                    $display("\n---Output is correct---");
                    $display("SPI_out = %b, SPI_Expected = %b\n", DUT_out,expected_tb);
                    end
                else begin
                    $display("Error!!!\n---Output is incorrect---");
                    $display("SPI_out = %b, SPI_Expected = %b\n", DUT_out,expected_tb);
                    $stop;
                end
            end
        endtask

    /*--------Self_Checking 8-bit---------*/
       task self_checking_8bit;
            input [7:0]DUT_out;
            input [7:0]expected_tb;
            begin
                // Check if the output is correct
                if (DUT_out == expected_tb) begin
                    $display("\n---Output is correct---");
                    $display("SPI_out = %x, SPI_Expected = %x", DUT_out,expected_tb);
                    end
                else begin
                    $display("\nError!!!\n---Output is incorrect---");
                    $display("SPI_out = %x, SPI_Expected = %x", DUT_out,expected_tb);
                    $stop;
                end
            end
        endtask

    endmodule //SPI_Master_tb
```

5. Do file to run testbench

```
#create Work Folder
vlib work

#Compile files with names
vlog Single_Port_Ram.v  SPI_Slave.v  SPI_Wrapper.v SPI_Master_tb.v

#simulate The TB file with module Name
vsim -voptargs=+acc work.SPI_Master_tb

#add the variables and internal signals with specific order to notice them easily

add wave -position insertpoint  \
sim:/SPI_Master_tb/MEM_DEPTH \
sim:/SPI_Master_tb/ADD_SIZE \
sim:/SPI_Master_tb/clk \
sim:/SPI_Master_tb/arst_n \
sim:/SPI_Master_tb/SS_n \
sim:/SPI_Master_tb/i \
sim:/SPI_Master_tb/data_addr_input \
sim:/SPI_Master_tb/DUT/SPI_slave/CS \
sim:/SPI_Master_tb/MOSI \
sim:/SPI_Master_tb/DUT/SPI_slave/rx_data \
sim:/SPI_Master_tb/DUT/SPI_slave/rx_valid \
sim:/SPI_Master_tb/DUT/RAM/addr_internal \
sim:/SPI_Master_tb/DUT/SPI_slave/tx_data \
sim:/SPI_Master_tb/DUT/SPI_slave/tx_valid \
sim:/SPI_Master_tb/MISO \
sim:/SPI_Master_tb/Data_output \
sim:/SPI_Master_tb/DUT/RAM/mem \


run -all

wave zoom full
```

6. Snippets from the waveforms

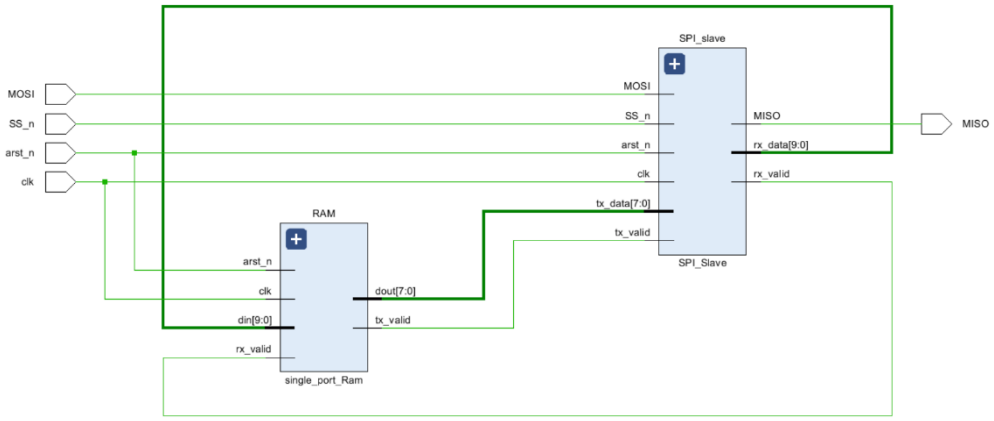


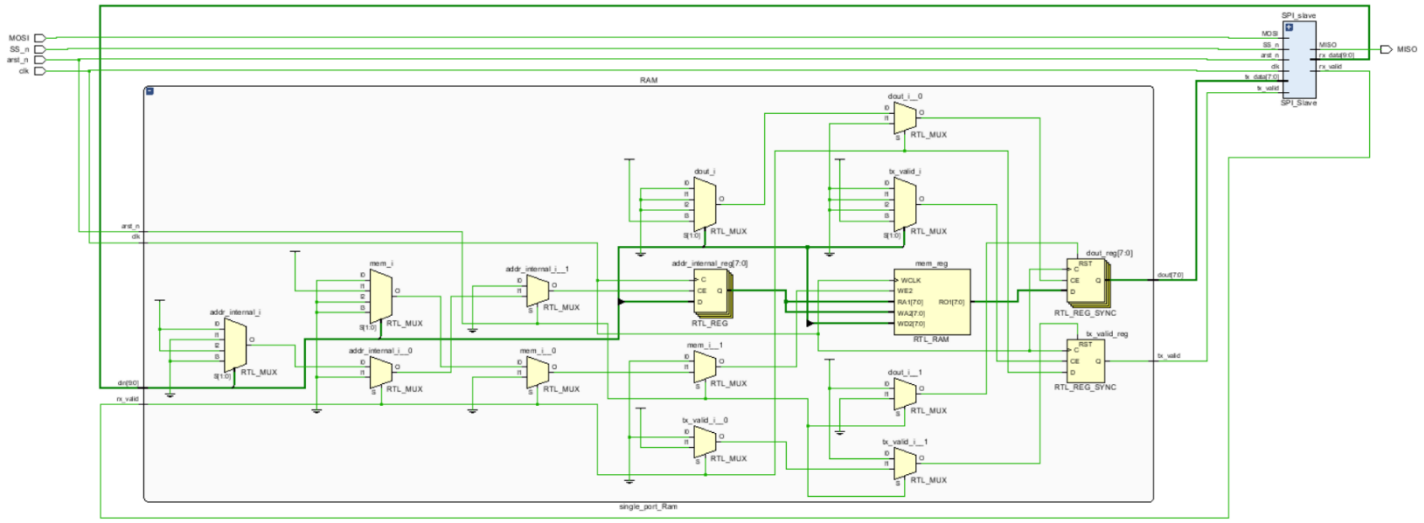
Showing the value stored in address “169”

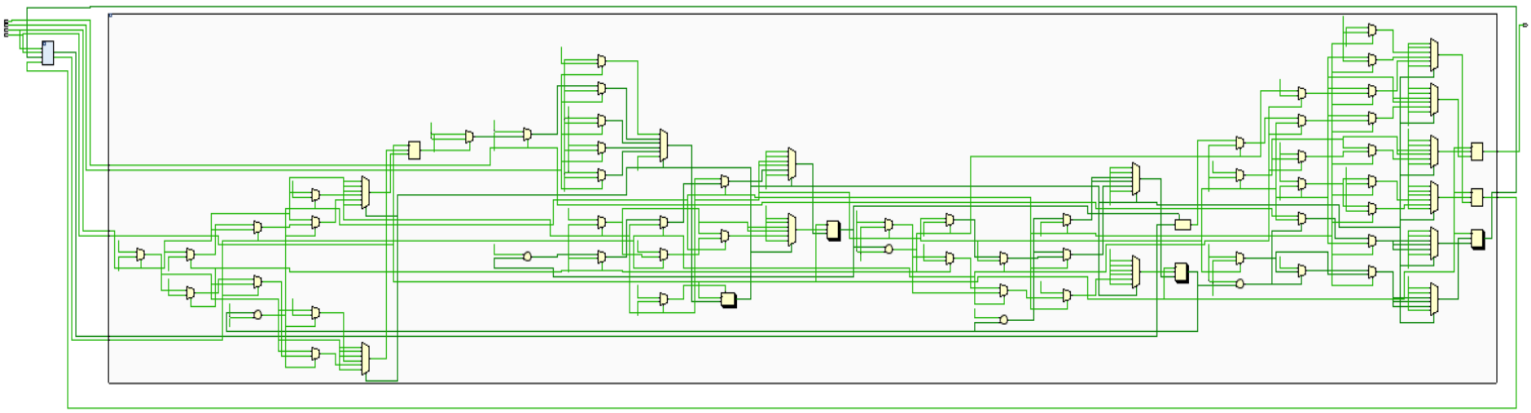
7. Constrains File after adding a debug core “sequential encoding”

```
## This file is a general .xdc for the Basys3 rev B board
## To use it in a project:
## - uncomment the lines corresponding to used pins
## - rename the used ports (in each line, after get_ports) according to the top level signal names in the project

## Clock signal
set_property -dict {PACKAGE_PIN W5 IOSTANDARD LVCMOS33} [get_ports clk]
create_clock -period 10.000 -name clk -waveform {0.000 5.000} -add [get_ports clk]

## Switches
set_property -dict {PACKAGE_PIN V17 IOSTANDARD LVCMOS33} [get_ports arst_n]
set_property -dict {PACKAGE_PIN V16 IOSTANDARD LVCMOS33} [get_ports SS_n]
set_property -dict {PACKAGE_PIN W16 IOSTANDARD LVCMOS33} [get_ports MOSI]
#set_property -dict { PACKAGE_PIN W17   IOSTANDARD LVCMOS33 } [get_ports {sw[3]}]

## LEDs
set_property -dict {PACKAGE_PIN U16 IOSTANDARD LVCMOS33} [get_ports MISO]
#set_property -dict { PACKAGE_PIN E19   IOSTANDARD LVCMOS33 } [get_ports {led[1]}]
#set_property -dict { PACKAGE_PIN U19   IOSTANDARD LVCMOS33 } [get_ports {led[2]}]
#set_property -dict { PACKAGE_PIN V19   IOSTANDARD LVCMOS33 } [get_ports {led[3]}]


##Buttons
#set_property -dict { PACKAGE_PIN U18   IOSTANDARD LVCMOS33 } [get_ports rst]

## Configuration options, can be used for all designs
set_property CONFIG_VOLTAGE 3.3 [current_design]
set_property CFGBVS VCCO [current_design]

## SPI configuration mode options for QSPI boot, can be used for all designs
set_property BITSTREAM.GENERAL.COMPRESS TRUE [current_design]
set_property BITSTREAM.CONFIG.CONFIGRATE 33 [current_design]
set_property CONFIG_MODE SPIx4 [current_design]

set_property MARK_DEBUG true [get_nets arst_n_IBUF]
set_property MARK_DEBUG true [get_nets clk_IBUF]
set_property MARK_DEBUG true [get_nets MISO_OBUF]
set_property MARK_DEBUG true [get_nets MOSI_IBUF]
set_property MARK_DEBUG true [get_nets SS_n_IBUF]
create_debug_core u_ila_0 ila
set_property ALL_PROBE_SAME_MU true [get_debug_cores u_ila_0]
set_property ALL_PROBE_SAME_MU_CNT 1 [get_debug_cores u_ila_0]
set_property C_ADV_TRIGGER false [get_debug_cores u_ila_0]
set_property C_DATA_DEPTH 1024 [get_debug_cores u_ila_0]
set_property C_EN_STRG_QUAL false [get_debug_cores u_ila_0]
set_property C_INPUT_PIPE_STAGES 0 [get_debug_cores u_ila_0]
set_property C_TRIGIN_EN false [get_debug_cores u_ila_0]
set_property C_TRIGOUT_EN false [get_debug_cores u_ila_0]
set_property port_width 1 [get_debug_ports u_ila_0/clk]
connect_debug_port u_ila_0/clk [get_nets [list clk_IBUF_BUFG]]
set_property PROBE_TYPE DATA_AND_TRIGGER [get_debug_ports u_ila_0/probe0]
set_property port_width 1 [get_debug_ports u_ila_0/probe0]
connect_debug_port u_ila_0/probe0 [get_nets [list arst_n_IBUF]]
create_debug_port u_ila_0 probe
set_property PROBE_TYPE DATA_AND_TRIGGER [get_debug_ports u_ila_0/probe1]
set_property port_width 1 [get_debug_ports u_ila_0/probe1]
connect_debug_port u_ila_0/probe1 [get_nets [list clk_IBUF]]
create_debug_port u_ila_0 probe
set_property PROBE_TYPE DATA_AND_TRIGGER [get_debug_ports u_ila_0/probe2]
set_property port_width 1 [get_debug_ports u_ila_0/probe2]
connect_debug_port u_ila_0/probe2 [get_nets [list MISO_OBUF]]
create_debug_port u_ila_0 probe
set_property PROBE_TYPE DATA_AND_TRIGGER [get_debug_ports u_ila_0/probe3]
set_property port_width 1 [get_debug_ports u_ila_0/probe3]
connect_debug_port u_ila_0/probe3 [get_nets [list MOSI_IBUF]]
create_debug_port u_ila_0 probe
set_property PROBE_TYPE DATA_AND_TRIGGER [get_debug_ports u_ila_0/probe4]
set_property port_width 1 [get_debug_ports u_ila_0/probe4]
connect_debug_port u_ila_0/probe4 [get_nets [list SS_n_IBUF]]
set_property C_CLK_INPUT_FREQ_HZ 300000000 [get_debug_cores dbg_hub]
set_property C_ENABLE_CLK_DIVIDER false [get_debug_cores dbg_hub]
set_property C_USER_SCAN_CHAIN 1 [get_debug_cores dbg_hub]
connect_debug_port dbg_hub/clk [get_nets clk_IBUF_BUFG]

```

8. Elaboration Schematic





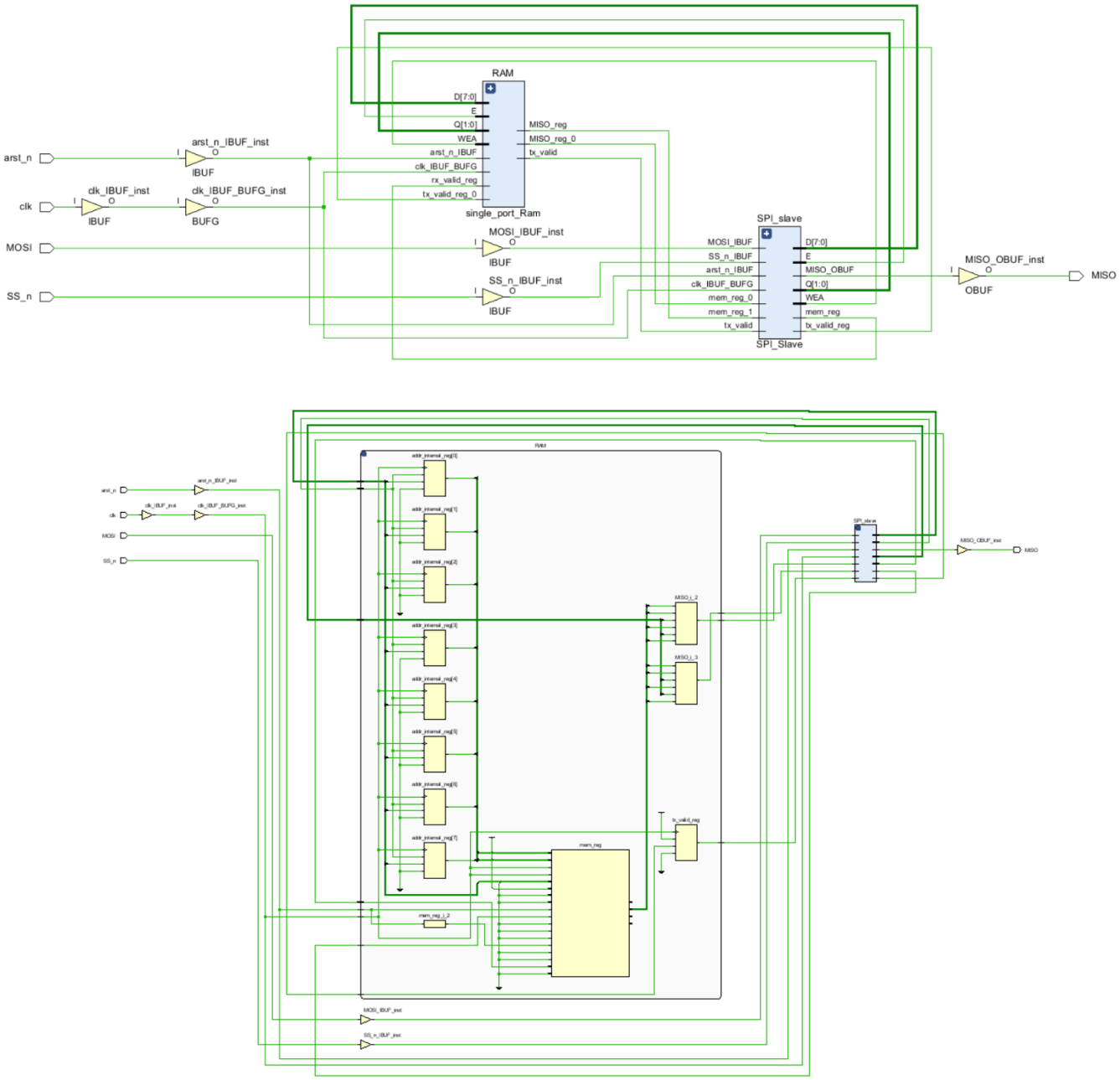


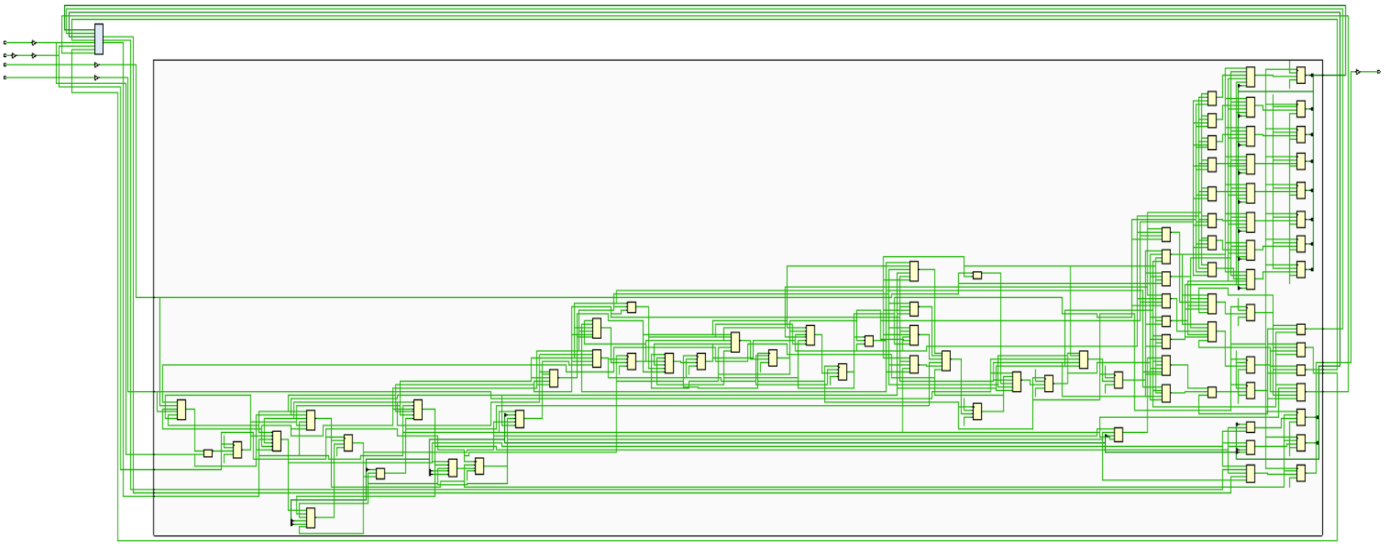
9. Synthesis Snippets for each encoding

9.1. Gray Encoding

```
-------------------------------------------------------------------------------------------------------
                State |                     New Encoding |                Previous Encoding
-------------------------------------------------------------------------------------------------------
                 IDLE |                              000 |                             000
              CHK_CMD |                              001 |                             001
                WRITE |                              011 |                             010
            READ_DATA |                              010 |                             100
             READ_ADD |                              111 |                             011
-------------------------------------------------------------------------------------------------------
INFO: [Synth 8-3354] encoded FSM with state register 'CS_reg' using encoding 'gray' in module 'SPI_Slave'
-------------------------------------------------------------------------------------------------------
```

9.1.1. Synthesis Schematic





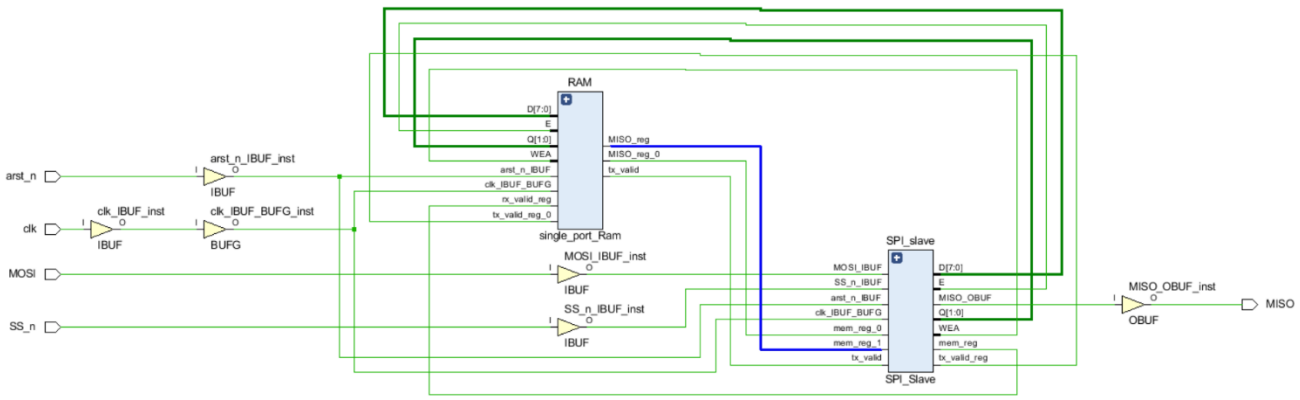
9.1.2. Timing Report summary

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	5.445 ns	Worst Hold Slack (WHS):	0.146 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	77	Total Number of Endpoints:	77	Total Number of Endpoints:	36

All user specified timing constraints are met.

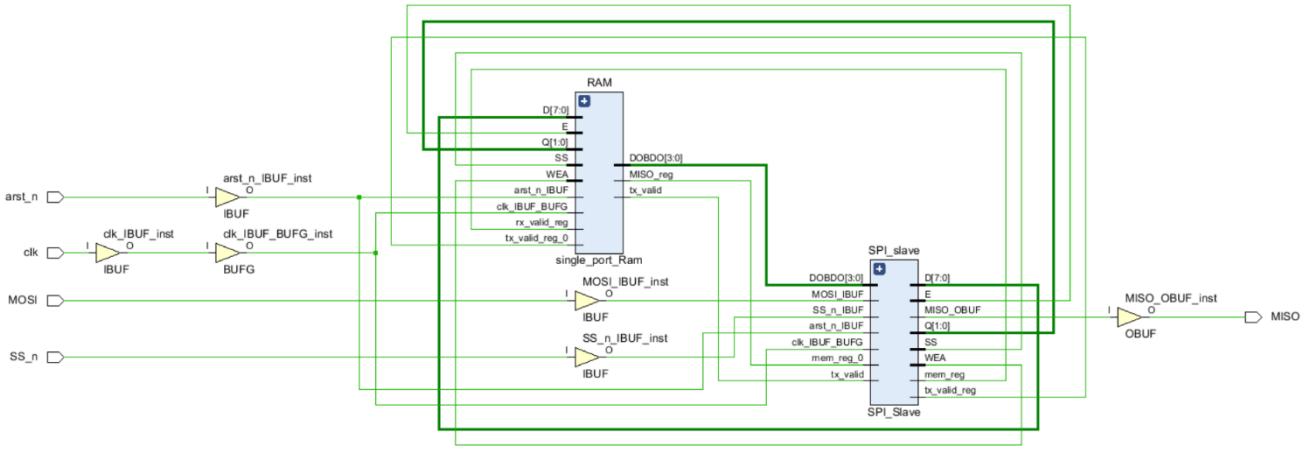
9.1.3. The critical path highlighted in the schematic

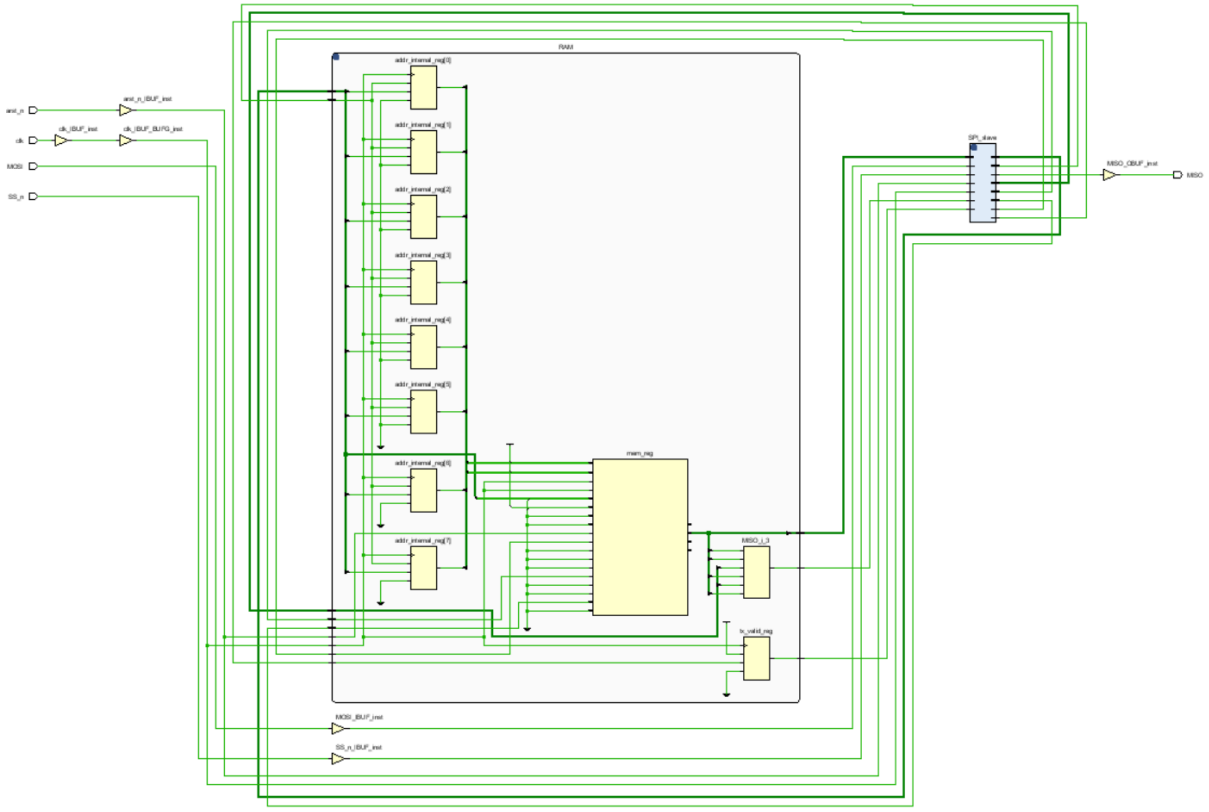


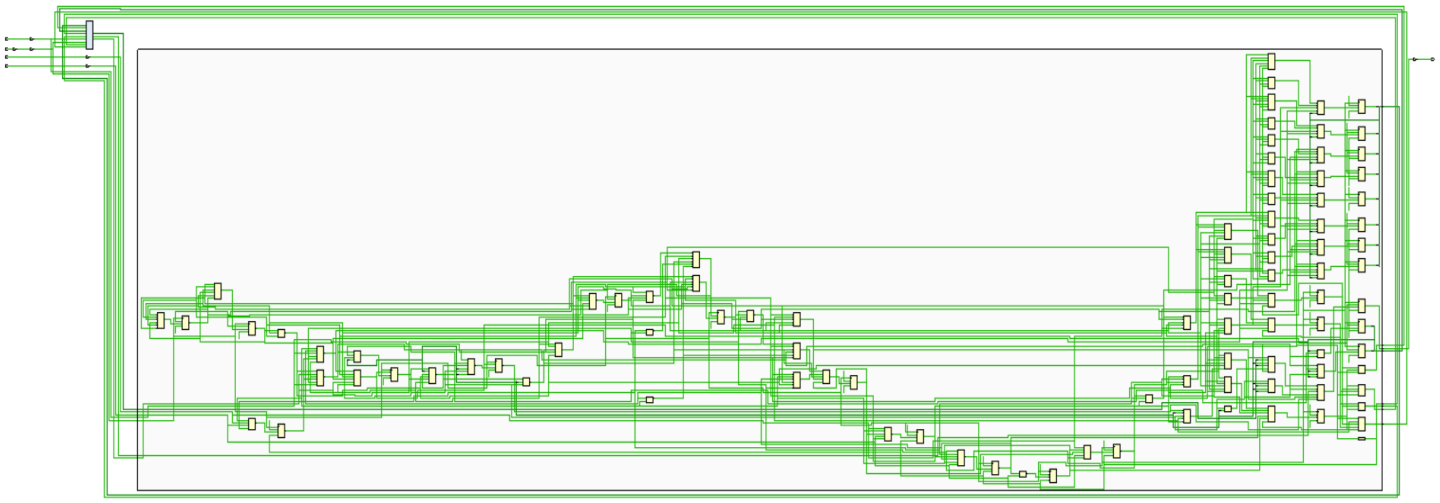
9.2. One_hot Encoding

```
-----------------------------------------------------------------------------------------------------
                State |                     New Encoding |                Previous Encoding
-----------------------------------------------------------------------------------------------------
                 IDLE |                            00001 |                             000
              CHX_CMD |                            00010 |                             001
                WRITE |                            00100 |                             010
            READ_DATA |                            01000 |                             100
             READ_ADD |                            10000 |                             011
-----------------------------------------------------------------------------------------------------
INFO: [Synth 8-3354] encoded FSM with state register 'CS_reg' using encoding 'one-hot' in module 'SPI_Slave'
-----------------------------------------------------------------------------
```

9.2.1. Synthesis Schematic







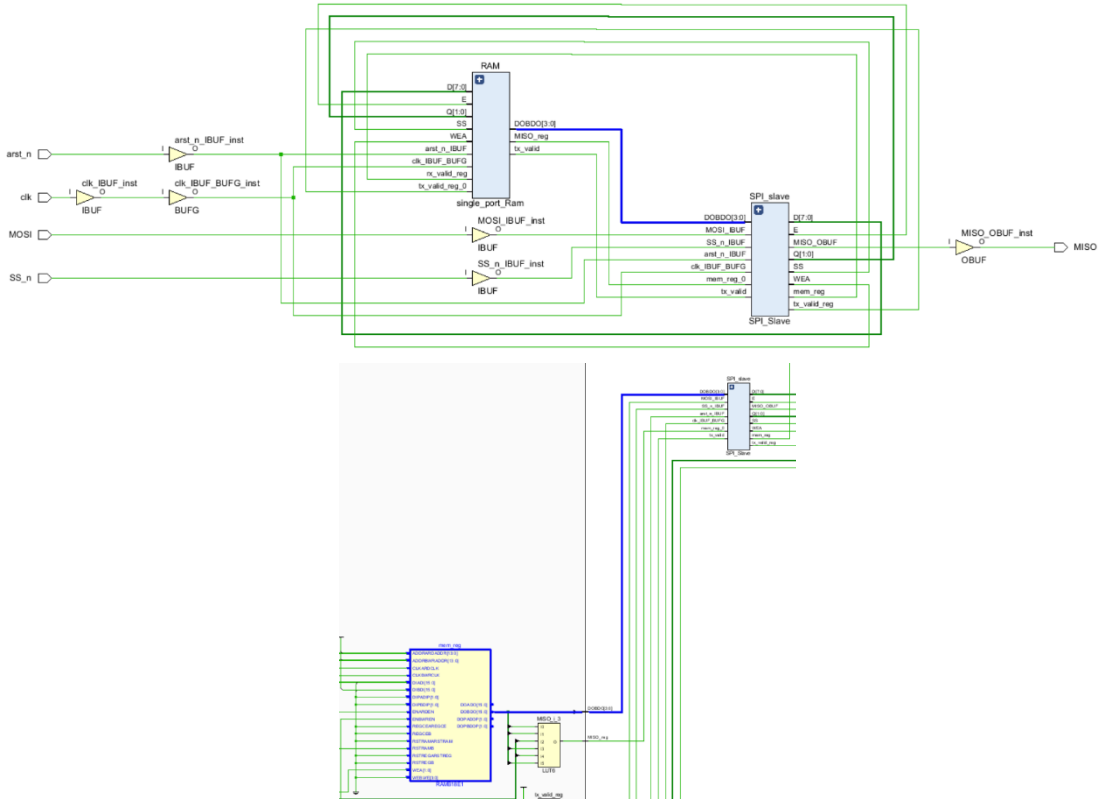
9.2.2. Timing Report summary

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	5.419 ns	Worst Hold Slack (WHS):	0.146 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	79	Total Number of Endpoints:	79	Total Number of Endpoints:	38

All user specified timing constraints are met.

9.2.3. The critical path highlighted in the schematic

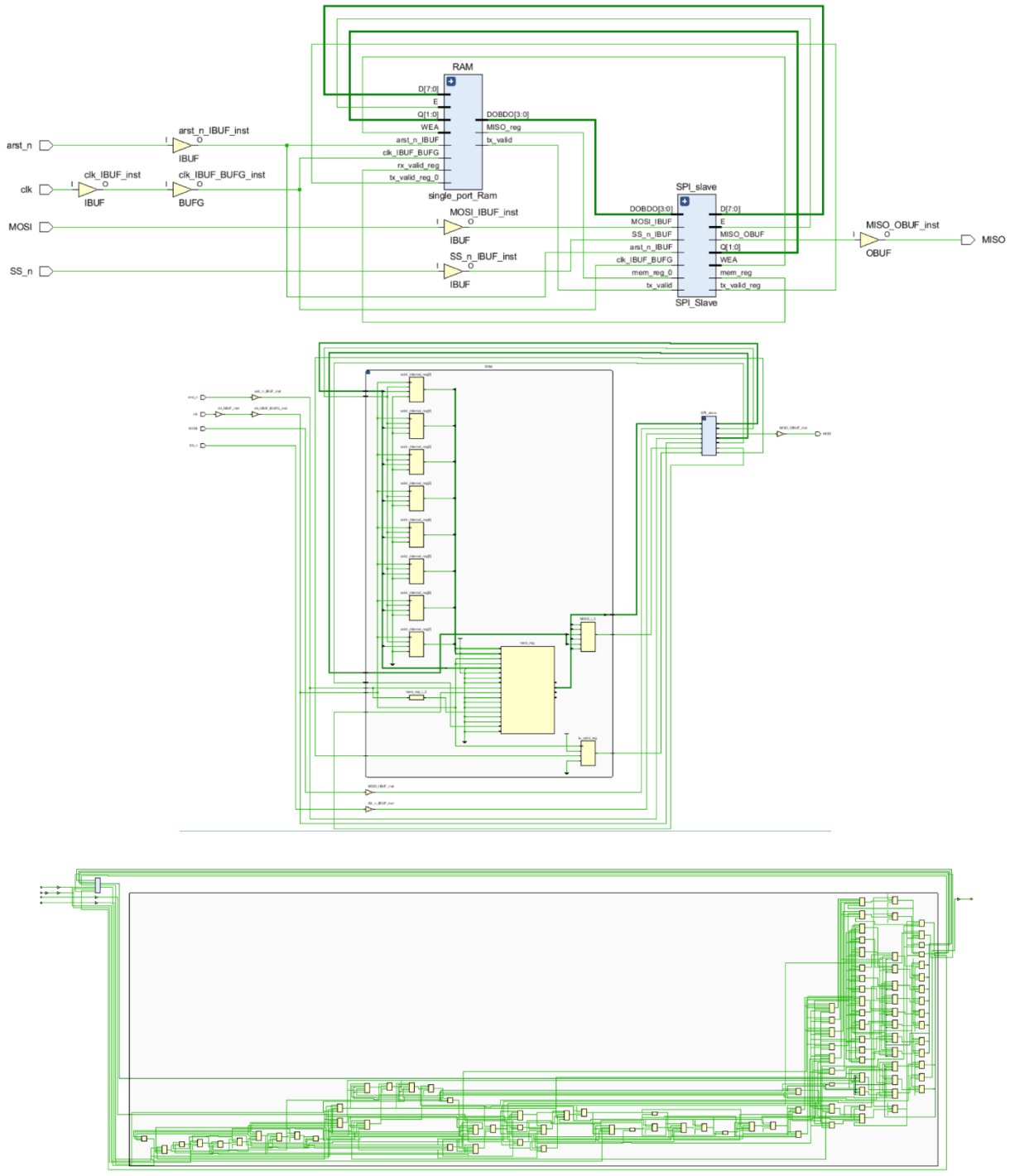


9.3. Sequential Encoding

State	New Encoding	Previous Encoding
IDLE	000	000
CHX_CMD	001	001
WRITE	010	010
READ_DATA	011	100
READ_ADD	100	011

: [Synth 8-3354] encoded FSM with state register 'CS_reg' using encoding 'sequential' in module 'SPI_Slave'

9.3.1. Synthesis Schematic



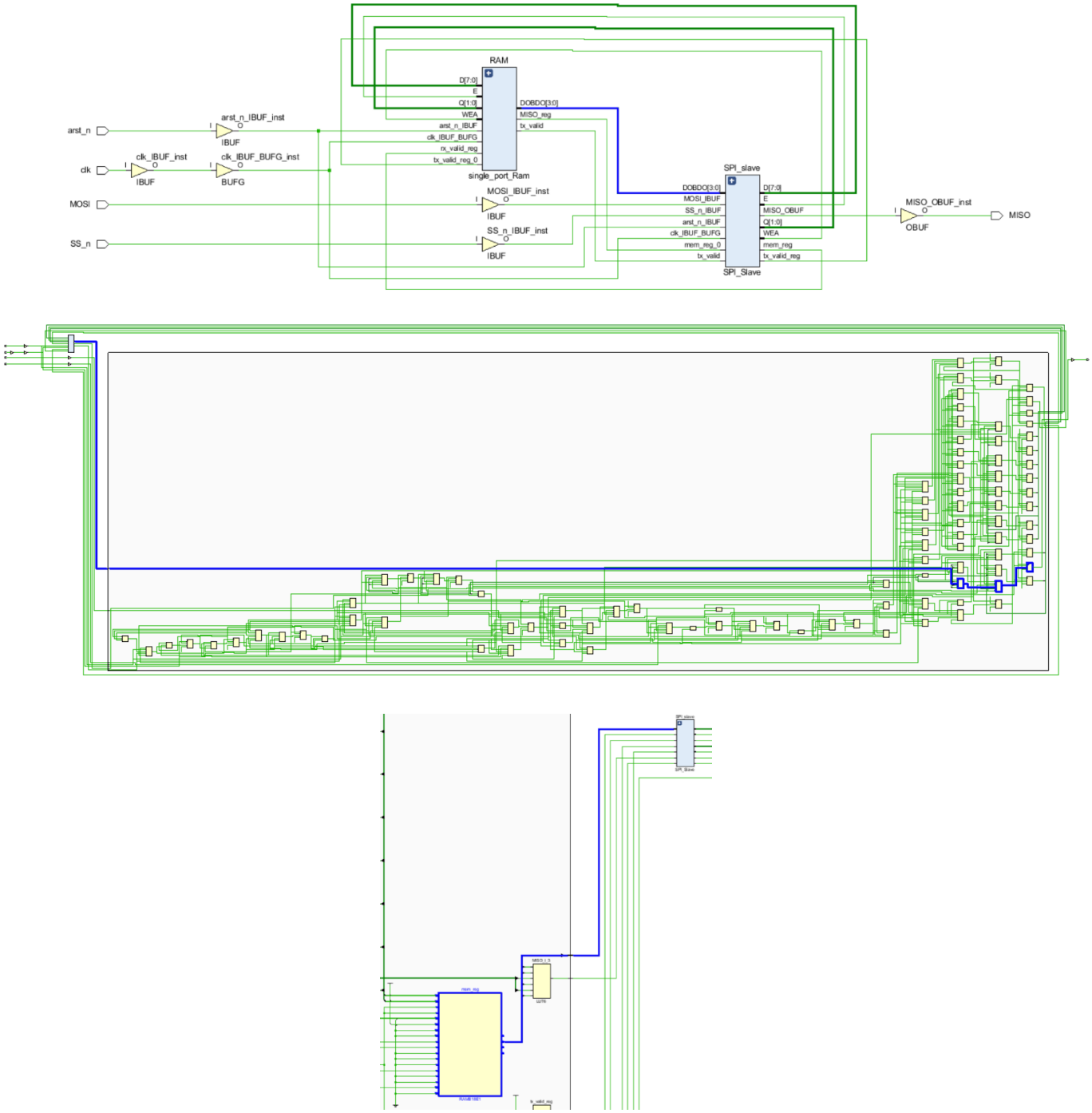
9.3.2. Timing Report summary

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	5.419 ns	Worst Hold Slack (WHS):	0.146 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	77	Total Number of Endpoints:	77	Total Number of Endpoints:	36

All user specified timing constraints are met.

9.3.3. The critical path highlighted in the schematic



10. Implementation snippets for each encoding

10.1. Gray Encoding

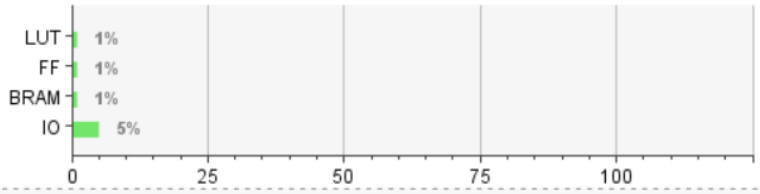
10.1.1. Utilization report

Hierarchy

Name	Slice LUTs (20800)	Slice Registers (41600)	Slice (8150)	LUT as Logic (20800)	LUT Flip Flop Pairs (20800)	Block RAM Tile (50)	Bonded IOB (106)	BUFGCTRL (32)
SPI_Wrapper	55	33	19	55	24	0.5	5	1
RAM (single_port_Ram)	3	9	5	3	0	0.5	0	0
SPI_slave (SPI_Slave)	52	24	17	52	23	0	0	0

Summary

Resource	Utilization	Available	Utilization %
LUT	55	20800	0.26
FF	33	41600	0.08
BRAM	0.50	50	1.00
IO	5	106	4.72



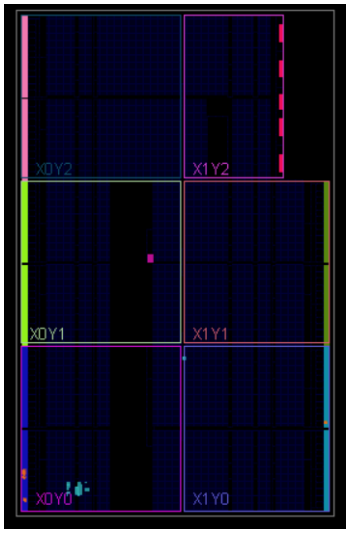
10.1.2. Timing report snippet

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	5.201 ns	Worst Hold Slack (WHS):	0.058 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	78	Total Number of Endpoints:	78	Total Number of Endpoints:	36

All user specified timing constraints are met.

10.1.3. FPGA device snippet



10.2. One_hot Encoding

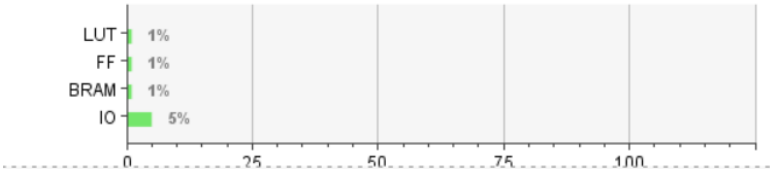
10.2.1. Utilization report

Hierarchy

Name	Slice LUTs (20800)	Slice Registers (41600)	Slice (8150)	LUT as Logic (20800)	LUT Flip Flop Pairs (20800)	Block RAM Tile (50)	Bonded IOB (106)	BUFGCTRL (32)
N SPI_Wrapper	64	35	19	64	26	0.5	5	1
RAM (single_port_Ram)	2	9	3	2	0	0.5	0	0
SPI_slave (SPI_Slave)	62	26	18	62	25	0	0	0

Summary

Resource	Utilization	Available	Utilization %
LUT	64	20800	0.31
FF	35	41600	0.08
BRAM	0.50	50	1.00
IO	5	106	4.72



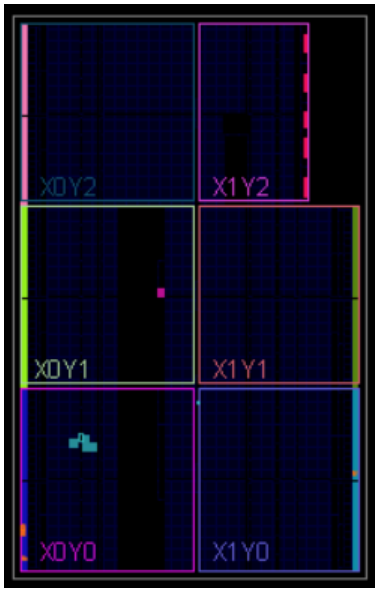
10.2.2. Timing Report summary

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	5.281 ns	Worst Hold Slack (WHS):	0.110 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	80	Total Number of Endpoints:	80	Total Number of Endpoints:	38

All user specified timing constraints are met.

10.2.3. FPGA device snippet



10.3. Sequential Encoding

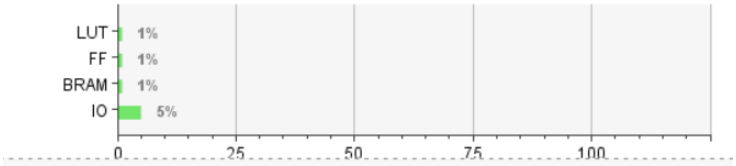
10.3.1. Utilization report

Hierarchy

Name	Slice LUTs (20800)	Slice Registers (41600)	Slice (8150)	LUT as Logic (20800)	LUT Flip Flop Pairs (20800)	Block RAM Tile (50)	Bonded IOB (106)	BUFGCTRL (32)
SPI_Wrapper	64	33	20	64	24	0.5	5	1
RAM (single_port_Ram)	2	9	3	2	0	0.5	0	0
SPI_slave (SPI_Slave)	62	24	20	62	23	0	0	0

Summary

Resource	Utilization	Available	Utilization %
LUT	64	20800	0.31
FF	33	41600	0.08
BRAM	0.50	50	1.00
IO	5	106	4.72



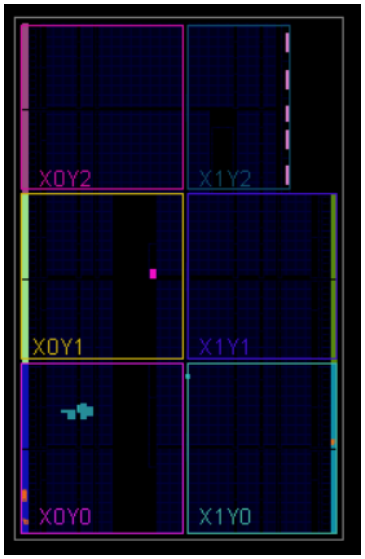
10.3.2. Timing Report summary

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	5.403 ns	Worst Hold Slack (WHS):	0.056 ns	Worst Pulse Width Slack (WPWS):	4.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	78	Total Number of Endpoints:	78	Total Number of Endpoints:	36

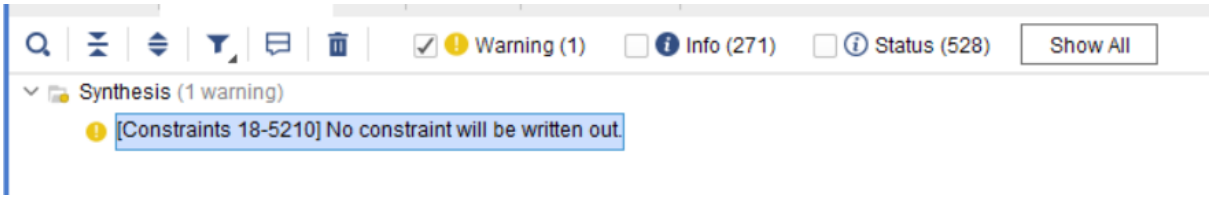
All user specified timing constraints are met.

10.3.3. FPGA device snippet



11. Snippet of the “Messages” tab

11.1. Gray Encoding



11.2. One_hot Encoding

11.3. Sequential Encoding